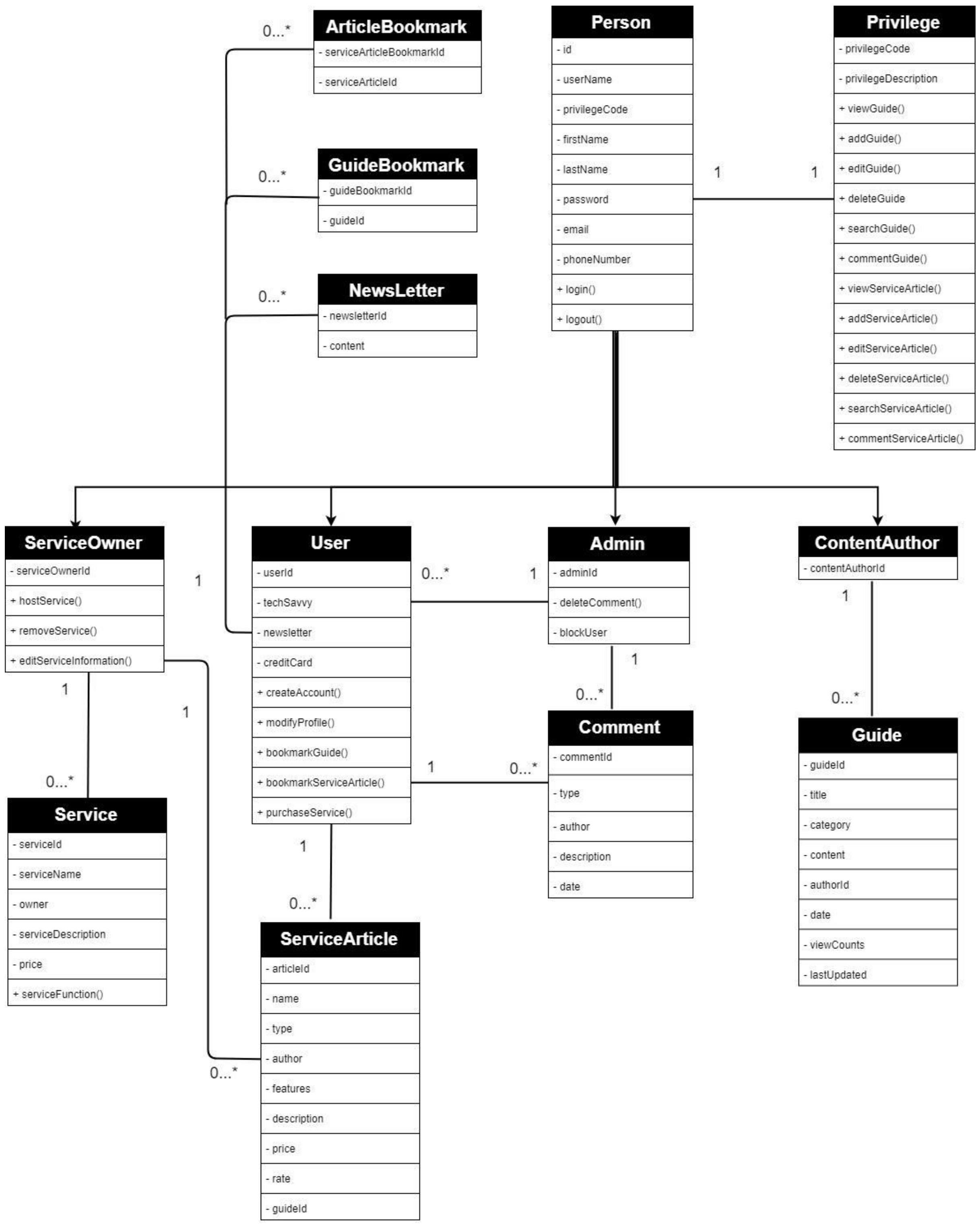
CS361 - Group 6 - HW4: Design Assignment
Sheng Bian | Michael Waldrop | Connor Shields | Linge Ge

UML class diagram with key OO entities



Entity Implementation Packaging

The OO design of our project can be implemented with low-coupling and high-cohesion. At the heart of our project is the Person class, from which every important type of user inherits. First, there is the User class. This is not a generic "User", but rather it is the User who interacts with the system without contributing. There are several important packaging schemes that relate to the User:

User-Newsletter:

Low degree of coupling. The Newsletter uses nothing more than the User ID to pass the data into the database. This involves passing structured data -- namely, the User ID from User to Newsletter -- but does not require any more involved interaction between the classes. If a change is made to the User class, it will likely not have any effect on the Newsletter class, because they are tied only through the User ID, a property which probably won't change when maintaining the system.

GuideBookmark-User-Service:

The same comments apply for the GuideBookmark class, because it only uses foreign keys to User and Service for it's generation of content. It is essentially just a structure with pointers to separate class entities, so it will rarely need to be updated when changes are made to the User and Service classes

Comment-User:

Modifications to the Comment class also has low-coupling and high-cohesion. Since the User and Comment classes both use the userID to pull information from the database, they have communicational cohesion. The User class uses the User ID to populate the User class variables, while the Comment class acquires necessary information about the user -- in this case, just the Author name -- to populate its own fields. Aside from that, their information is entirely dependent on their own classes.

Person-Privilege-Guide:

The general idea of the privilege class is that it uses the privilegeCode associated with each person to determine what types of functions it can perform. The permissions for performing each of these functions could be implemented using simple conditional statements. For instance, if a user's permissionID matches the permissionsID required to write a guide for a service, then the UI for that user will include a button to create a guide. This could be seen as an example of functional cohesion, because the guide is created and modified based on the permissions level of the user, indicating that both the userID and permissionID fields from two classes are working together to fulfill the same purpose of creating a guide. Again, any updates to the Person or Privilege classes won't carry much, if any, overhead for updating the Guide class, since it only relies on the conditional checking from the Privilege class.

<u>**Incremental and Iterative Development**</u>

Our design supports incremental and iterative development very well. In order to support incremental development, much of the system's value should reside in one subsection and one part of the system must be completed before. First, the whole system's value is to help people protect their privacy. Then, guide and technology service are two most important subsections in our system. Users can learn useful information about how to protect privacy by reading guides and use a variety of technology services to protect their privacy online. Second, in our system, we need to create "Person" module first, then we can create other roles like user, admin and content author. After that, we can assign different privilege for different roles. Another example is we need to build the main system first before we add a newsletter module. Besides, the bookmark system also needs to be added later after we build the basic system.

In order to support iterative development, the system's value should be spread out over much of the system and the whole system needs to work at least a bit before we can build up. In our system, we will first build "Person" class. "User", "Admin" and "ContentAuthor" will inherit from "Person" class. These three different "Person" have different privilege for the system. After those roles are working, we can add specific functions for specific roles. If we want to change "Person" attributes, the changes are spread across three roles. Besides, "comment" subsystem is open to three roles and accessible by both guide and technology service. The changes and improvements to "comment" subsystem will also spread out over the system. The guide and technology service are two most important pieces of the system. We can make improvements to the user interface of those two sections and it will improve the usability of the whole system.

<u>**Proposed Design Patterns**</u>

In regards to design patterns for our project, there were several which were suitable for different reasons. These different patterns are listed below, along with the reasoning for each:

<u>Façade</u>

- This interface may be appropriate, because the project is primarily a website with clickables that activate different algorithms. The user is initially presented with a home page that allows them to choose between four different options: Create an account, login, search guides, and search services. Each of these options activates a different function under the simple premise of clicking a button on the user's end. The functions are as follows:
    - **Create Account**: Leads to a form fill-in and submission that also leads to account validation emails and an automatic login once the process is complete. Even as the user moves through the account creation process, they are not involved in the activation of the different functions beyond filling information into a form and clicking a button a few times.

- **Login**: This requires a simple check against the database for an existing user, and proper password to validate the username credential. The act of logging in also unlocks additional functionality for the user, and redirects them to a page they would not be able to access without having an account. This is all provided by filling in two lines and clicking a button.
- **Search Guides/Services**: Clicking each of these will allow the user to see an automatically populated list, or refine their search based on information they enter or filters they set. Searching and filtering are more complicated than the handful of clicks it would require for the user to activate them, so a more complicated functionality is hidden behind a fairly simple user interface.
- At any time the user can navigate around the website with the click of a button, or reactivate the functions specified above by navigating back to the specific page for that function or accessing the buttons provided in each screen. The interface itself is a fairly simplistic means to repeatedly access more complex functionality.
- In addition, the user review and rating system allows for users to add new entries into the database or modify existing entries (to a limited extent, unless they are actually editing a guide). While adding an entry into a database is not particularly complex in comparison to the form filling needed to create the entry, this could still be considered a simplified UI hiding a more complicated function.

<u>Adaptive</u>

- Adaptive could be relevant for this project, because of the need to switch between interfaces that do not allow exactly the same, or even similar options. For example, simply accessing the database through a pre-populated list of bookmarks is not the same as actively searching for new entries, or adding new entries to the database. The overall intention of this design pattern, removing incompatibility, does not reach its full potential with this project, however.

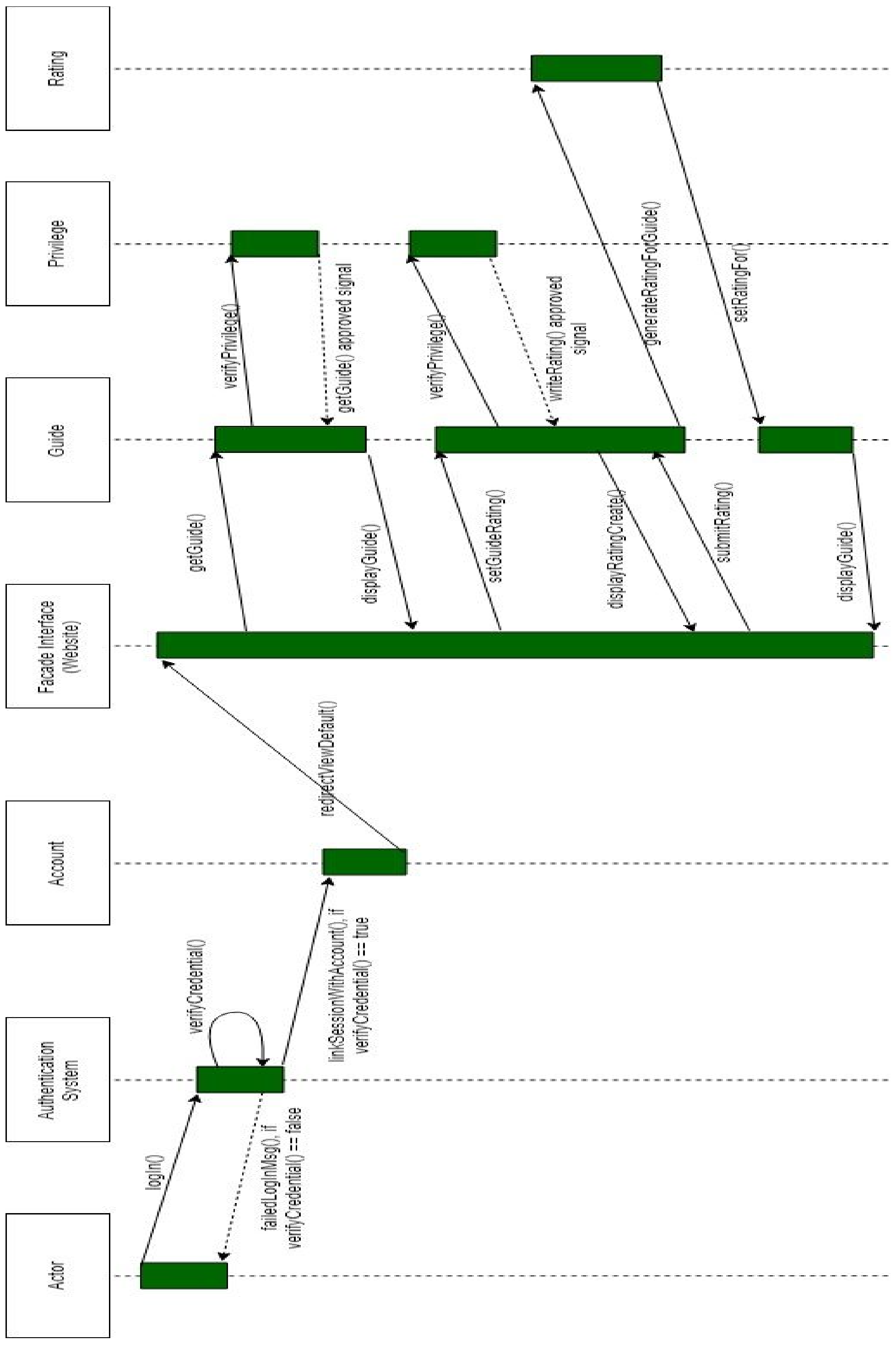
<u>Observer</u>

- Observer could be relevant for this project purely because the act of creating an account serves as a gate to block users with less than a certain level of authority from actively modifying the database. Once users have validated their account, the “permissions” observers would allow them access to the add/edit functions, as well as the bookmarking functionality. The project could be considered event-driven, because accessing certain functions requires the user to activate and pass the Create Account and Login events. In a general sense, access to the database cannot happen unless the user activates the Search events from the homepage, as information is not provided without actually moving further into the website.

<u>Visitor</u>

- Similar to the observer explanation, Visitor could be relevant because both verified and unverified users can access some of the same functionality: search. However, the algorithm for search is slightly different for verified users, as it allows them the additional ability to add entries to the database or add reviews for existing entries. In this sense, both types of user can access the Search algorithms, but only verified users can access the “Add/Edit” function of the database algorithm.

Use Case Sequence Diagram (#2):
User logins in successfully, views a guide, then creates a rating for the same guide. Primary interactions involve user interacting with a Facade which sends request methods to get and interact with instances of the following classes: Guide, Privilege, Rating.



New Use Case #3

Name: Service owner hosts service on the website

Actor: Service owner

Preconditions:

- Service owner has created the service for protecting user privacy
- Service owner has an account for the website
- The service can be hosted on the website

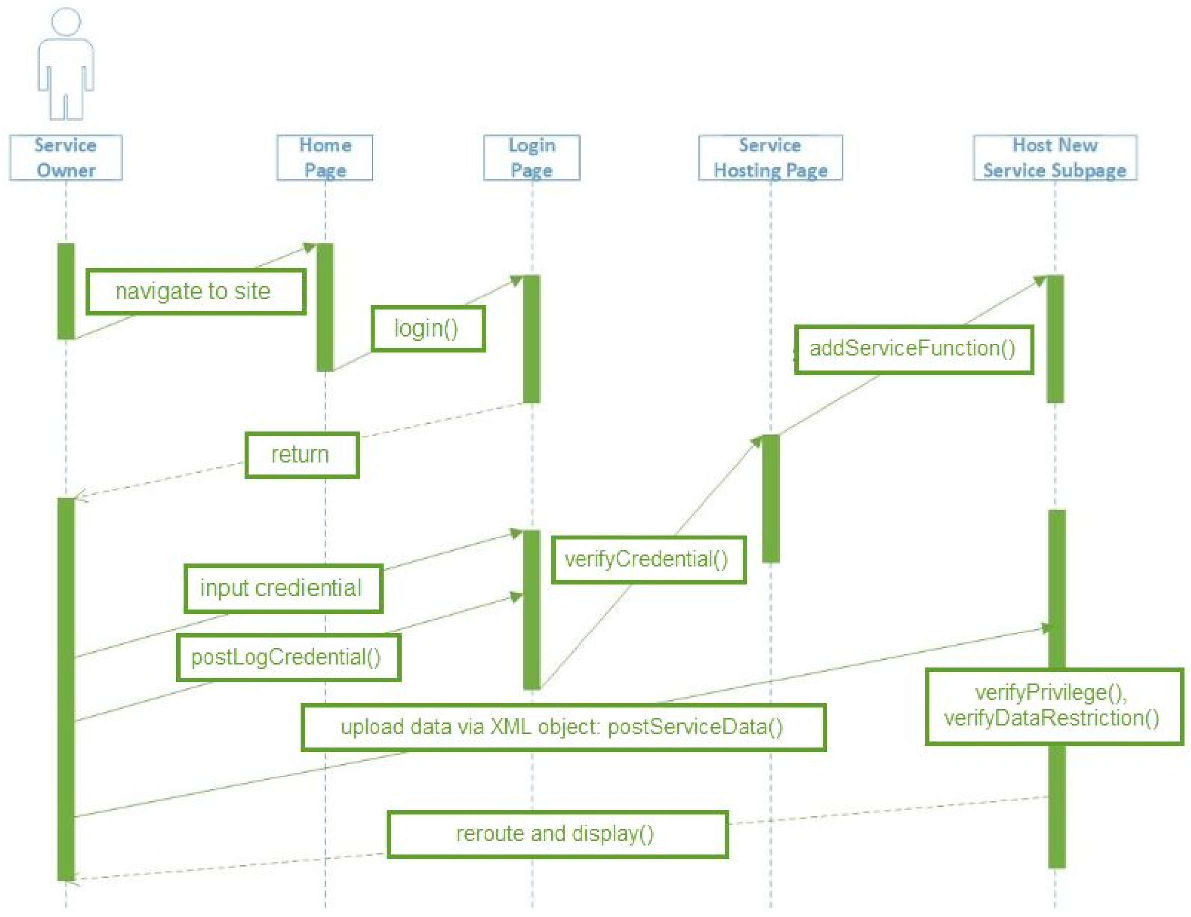
Postconditions:

- Service owner successfully hosts service on the website
- User can subscribe the hosted service on the website

Flow of Events:

1. Service owner navigates to the website home page.
2. Service owner clicks login at the top right of the web page.
3. A text entry box opens
4. Service owner selects “Log in as Service Owner”.
5. Service owner is prompted to enter account information and clicks submit to log in.
6. Service owner is presented with a service hosting page.
7. Service owner selects “Host New Service”.
8. Service owner is taken to “Host New Service” subpage where service owner can host the new service
9. Service owner selects “Upload Code of New Service” and select code from local computer.
10. Service owner is prompted enters new service information, including service name, service description and price.
11. After uploading code and filling out all the information, service owner clicks submit.
12. System checks if the new service can be successfully hosted.
13. Service owner is presented the result message of hosting new service.
    a) “You have successfully hosted new service!”
    b) “Error! New service wasn’t hosted successfully!”

<u>Use Case #3 - Message Sequence Diagram</u>



<u>Interfaces</u>:

Each of the interfaces in our system requires a condition. The user's permission level is the only variable used to satisfy these conditions. Namely, there are three classes that inherit from the Person interface: User, Admin, and contentAuthor

- The User is the most basic of the classes, as this is the type of person who can search the website, read service guides, bookmark pages, and write comments. No actual permission level is required for a person to become a user. A person visiting the site is a user by default, meaning that no conditional checking is performed to allow the user to access the parts of the website which a user can interact with.
- Admin, by contrast, is the Person who has the most permissions. A person can only access features with admin credentials if their privilege Code matches a range of privilege Codes that could represent an admin user. In order for a user to gain an admin privilege code, they must have an admin id. The privilege code is generated based on

the status of the user ID passed into that class (there are unique formats for admin, user, and contentAuthor IDs. The ID might just be a string of digits with a different letter in front to represent the Person type: ‘A’ for admin, ‘U’ for user, and ‘CA’ for contentAuthor). If the adminID matches an admin ID in the database, then the privilege code generated for that person is also one that gives them admin privilege. This is checked again when trying to perform certain actions on the website. To block a user or delete a comment, the function for performing either action checks whether or not the privilege code is valid. If an admin user has the proper privilege code, as they should, they will be able to trigger the functions on the content.

- The contentAuthor is the last class type that inherits from the Person interface. Objects of this type represent users who have an author ID, which, just like the admin ID, allows them to gain a privilege code that allows them to use functions for manipulating content. A contentAuthor can search guides and services just like a regular User, but they can also add guides to services as they see fit. A contentAuthor can also make edits to guides that they have written. This constraint is checked by validating the user ID associated with the service guide against the contentAuthor’s author ID. If it is the same, or if they are an admin, then they can edit the guide. Otherwise, they cannot.

Exception and Exception Handlers

Several exceptions are likely to occur within the system, mostly due to concurrent user interaction since most pages of the website interface is static:

- User adds a bookmark to a guide which was deleted moments before while user was still on the same page.
    - Handler: verify existence of guide at time of bookmark addition to catch exception, negate the usual method execution and do nothing, and throw a friendly user warning.
- User submits a review for guide/service that was removed while user was still on the same page.
    - Handler: Anytime a guide/service is deleted, find users that are currently rating the guide/service, and throw a friendly warning and render the page submit rating button non-operational.
    - Note: this isn’t exactly an exception handler, but more of a preemptive prevention to prevent the exception from ever being reached.
- A service is offered that no longer has a linked Guide ID due to guide deletion.
    - Handler: links to deleted guide(s) in the service should automatically be wiped
- An admin deletes a rating on a service/guide which has already been deleted.
    - Handler: verify existence of rating at time of deletion to catch exception, negate the usual method execution and do nothing, and throw a friendly user warning.
- User uploads a profile picture that does not meet system requirements.
    - Handler: Catch failed requirements met, negate upload, and throw friendly user warning.
- User attempts to access a guide/service from search which has been deleted after search page generation.

- Handler: Verify the existence of guide/service, and if it does not exist, then stop navigation to that guide/service, allow user to stay on same search page, and throw friendly warning.

Weekly Work Summary

- Sheng drew the UML class diagram showing all of the various OO entities in our expected product. He also stepped through the expected incremental and iterative development of the product based on the existing requirements.
- Connor created the packaging scheme for the implementation of the entities in the UML diagram, and assessed the packaging scheme in terms of coupling and cohesion. Connor also listed the interfaces that would be needed and gave contracts for each.
- David created updated sequence diagrams using a combination of use cases #2 and #3 from the earlier assignments with an expected Facade design pattern guiding the primary interactions. David also identified and listed the exceptions that were likely to occur with our planned system.
- Michael presented a list of design patterns that would be applicable to the system, and provided reasoning for each. Michael also wrote the work summary detailing the contributions of each group member for the week.